Java - Funktionale Programmierung

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de , Raum 149B
Version: 2.0.2.1

Teile der Folien basieren auf: Th. Letschert - Funktionale Programmierung in Java.

Kontrollfragen: kontrollfragen.de.md.html
Folien: https://delors.github.io/prog-adv-java-funktionale-programmierung/folien.de.md.html
https://delors.github.io/prog-adv-java-funktionale-programmierung/folien.de.md.html.pdf
Fehler melden: https://github.com/Delors/delors.github.io/issues
KI Verwendung: Bei der Erstellung der Unterlagen wurden KI Assistenten (insbesondere Claude aber ggf. auch ChatGPT, Ollama mit Gemma/LLama/Qwen oder OpenCode mit Kimi/Qwen/Deepseek...) unterstützend eingesetzt. Dies erfolgte insbesondere zur Unterstützung bei der Generierung von Grafiken (d. h. SVG Dateien), oder um sich Übersichtstabellen generieren zu lassen. Weiterhin wurde KI zur allgemeinen Qualitätssicherung eingesetzt. Inhalte, die ggf. von der KI vorgeschlagen wurden, wurden im Falle der Übernahme explizit validiert und angepasst.

1. Einführung in die Funktionale Programmierung

Grundlagen funktionaler Programmierung

- Programmierparadigma, bei dem Funktionen im Mittelpunkt stehen
- Vermeidet veränderliche Zustände (🇺🇸 *Mutable State*)
- Fördert deklarativen Code statt imperativem Code

? Frage

Wie unterscheidet sich dieses Paradigma von der objektorientierten Programmierung?

✓ Antwort

- Methoden ohne Seiteneffekte
- Daten sind standardmäßig unveränderlich
- Fokus auf Funktionsanwendungen und -komposition

Wichtige Konzepte

- Funktionen Höherer Ordnung
- Lambda-Ausdrücke
- Funktionskomposition
- Currying und Partielle Anwendung

Beim Currying wird eine Funktion mit n Argumenten in eine Kette von n Funktionen umgewandelt, die jeweils genau ein Argument entgegennehmen: f(a,b,c) wird zu f(a)(b)(c).

Bei der partiellen Anwendung wird eine Funktion nur mit einem Teil ihrer Argumente aufgerufen. Das Ergebnis ist eine neue Funktion, die die weiteren Parameter entgegennimmt und erst dann die Originalfunktion aufruft.

2. Funktionale Programmierung in Java

Lambdas

Lambda (auch Closure):

Ein Ausdruck, dessen Wert eine Funktion ist.
Solche Ausdrücke sind sehr nützlich, mussten in Java aber vor Version 8 mit anonymen inneren Klassen emuliert werden.

Ein einfache Personenklasse

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
    public String toString() { return "Person[" + name + ", " + age + "]"; }
}
```

Sortieren von Personen nach Alter

Angenommen wir haben eine Klasse Person und eine Liste von Personen, die nach Alter sortiert werden soll. Dazu muss eine Vergleichsfunktion übergeben werden. In Java <8 kommt dazu nur ein Objekt in Frage.

```
List<Person> persons = Arrays.asList(
    new Person("Hugo", 55),
    new Person("Amalie", 15),
    new Person("Anelise", 32) );
```

Traditionelle Lösung

```
Collections.sort(persons, new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
});
```

Lösung ab Java 8+

```
Collections.sort(
    persons,
    (p1, p2) -> { return p1.getAge() - p2.getAge(); }
);
```

Lösung ab Java 8 (kürzer)

```
Collections.sort(persons, (p1, p2) -> p1.getAge() - p2.getAge());
```

▲ Achtung!

java.lang.`Object` ist zwar der Basistyp aller Referenztypen, der Typ eines Lambdas ist jedoch der Typ eines funktionalen Interfaces und dieser Typ muss explizit angegeben werden. Andernfalls wäre der Compiler nicht in der Lage den Typ aus dem Kontext zu ermitteln.

Funktionale Interfaces deklarieren genau eine abstrakte Methode. Für viele Standardfälle sind Interfaces in `java.util.function` vordefiniert.

Instanzen von anonymen inneren Klassen können immer Variablen vom Typ `java.lang.Object` zugewiesen werden:

```
Object actionListener = new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println(text);
    }
};
```

Illegale Zuweisung:

```
Object actionListener = (e) -> System.out.println(text);

// Error: The target type of this expression must be a functional interface
```

Zuweisung an ein funktionales Interface:

```
ActionListener actionListener = (e) -> System.out.println(text);
```

Selbstdefinierte funktionale Interfaces

Functional Interface / SAM-Interface (Single Abstract Method Interface):

Ein *Functional Interface* ist ein Interface das genau eine abstrakte Methode enthält.
Um die intendierte Verwendung explizit zu machen, kann optional die Annotation `@FunctionalInterface` hinzugefügt werden. In diesem Fall wird durch den Compiler auch die Einhaltung der Anforderungen an funktionale Interfaces überprüft.

Beispiel

```
@FunctionalInterface
interface MyActionListener extends java.awt.event.ActionListener {
    /*final static*/ int MAGIC_NUMBER = 42;
}

MyActionListener actionListener =
        (e) -> System.out.println(text + MyActionListener.MAGIC_NUMBER);
```

Vordefinierte Funktionsinterfaces

`java.util.function` enthält viele vordefinierte Funktionsinterfaces, die in der funktionalen Programmierung häufig verwendet werden.

Beispiele sind:

- `Function<T,R>`: Eine Funktion, die ein Argument vom Typ `T` entgegennimmt und ein Ergebnis vom Typ `R` zurückgibt.
- `Predicate<T>`: Eine Funktion, die ein Argument vom Typ `T` entgegennimmt und ein Ergebnis vom Typ `boolean` zurückgibt.
- `Consumer<T>`: Eine Funktion, die ein Argument vom Typ `T` entgegennimmt und kein Ergebnis zurückgibt.
- `Supplier<T>`: Eine Funktion, die kein Argument entgegennimmt und ein Ergebnis vom Typ `T` zurückgibt.

Beispiel

```
Predicate<T>
```

```java
static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
    List<T> res = new LinkedList<>();
    for (T x : l) {
        if (pred.test(x)) { res.add(x); }
    }
    return res;
}

List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
System.out.println(filterList(l, x -> x % 2 == 0));
```

Ausgabe: `[2, 4, 6, 8]`

Beispiel

`Consumer<T>`

```java
class WorkerOnList<T> implements Consumer<List<T>> {
    private Consumer<T> action;
    public WorkerOnList(Consumer<T> action) { this.action = action; }

    @Override public void accept(List<T> l) {
        for (T x : l) action.accept(x);
}   }

WorkerOnList<Integer> worker =
        new WorkerOnList<>( (i) -> System.out.println(i*10) );
worker.accept(Arrays.asList(1,2,3,4));
```

Ausgabe: `10 20 30 40`

Lambdas - Method References

Als Implementierung eines funktionalen Interfaces (als „Lambda“) können auch Methoden verwendet werden.

Beispiel

Referenz auf statische Methode

```
class ListMethods {
    static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
        List<T> res = new LinkedList<>();
        for (T x : l) if (pred.test(x)) { res.add(x); }
        return res;
    }
    static boolean isEven(int x) { return x % 2 == 0; }
}

List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
System.out.println(filterList(l, ListMethods::isEven));
```

Ausgabe: `[2, 4, 6, 8]`

Beispiel

Referenz auf Instanzmethode

```
class Tester {
    private int magicNumber;
    public Tester(int magicNumber) { this.magicNumber = magicNumber; }
    boolean isMagic(int x) { return x == magicNumber; }
}
class ListMethods {
    static <T> List<T> filterList(List<T> l, Predicate<T> pred) {
        List<T> res = new LinkedList<>();
        for (T x : l) if (pred.test(x)) res.add(x);
        return res;
}   }

List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
System.out.println(filterList(l, new Tester(5)::isMagic));
```

Beispiel

Referenz auf Konstruktor

```
class Tester {
    private int magicNumber;
    public Tester(int magicNumber) { this.magicNumber = magicNumber; }
    boolean isMagic(int x) { return x == magicNumber; }
}

Function<Integer,Tester> create = Tester::new;
create.apply(5).isMagic(5);
```

Ausgabe: `true`

Erweiterungen der Collection API

Neue Methoden in der Collection API

- `forEach(Consumer<? super T> action)`
- `removeIf(Predicate<? super T> filter)`
- `replaceAll(UnaryOperator<T> operator)`
- `sort(Comparator<? super T> c)`

Beispiel

```
1 List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
2 l.replaceAll(x -> x * 10);
3 l.forEach(System.out::println);
```

Ausgabe: `10 20 30 40 50 60 70 80 90`

✎ Übung - Lambda-Funktionen

2.1. Einfache Lambda-Funktionen

Implementieren Sie die beiden folgenden TODOs.

```
public static void main(String[] args) {
    List<String> names = Arrays.asList("Diana", "Alice", "Charlie", "Bob", "Eve");
    List<Integer> grades = Arrays.asList(85, 42, 91, 67, 55);

    // TODO 1: Using a lambda, sort the names
    names.sort(...);

    // TODO 2: Print the names to the console using forEach and a method reference
    names.forEach(...);
}
```

✎ Übung

2.2. Erste Implementierung von Funktionen höherer Ordnung

Schreiben Sie eine Klasse `Tuple2<T>`; d. h. eine Variante von `Pair` bei der beide Werte vom gleichen Typ `T` sein müssen. Die Klasse soll Methoden haben, um die beiden Werte zu setzen und zu lesen. Weiterhin soll die Klasse um folgende Methoden ergänzt werden:

- `void forEach(Consumer<...> action)`: Führt die Aktion für jedes Element in dem `Tuple` aus.
- `void replaceAll(UnaryOperator<...> operator)`: Ersetzt alle Elemente im `Tuple` durch das Ergebnis der Anwendung des Operators auf das Element.

Schreiben Sie Tests für die neuen Methoden; verwenden Sie dafür Closures bzw. Lambda-Funktionen. Stellen Sie 100% *Statementcoverage* sicher.

Die Java Dokumentation finden Sie hier:

- Übersicht: https://docs.oracle.com/en/java/javase/25/docs/api/help-doc.html
- API Dokumentation: https://docs.oracle.com/en/java/javase/25/docs/api/allclasses-index.html

  (Hier der Link auf die Dokumentation der Klasse `UnaryOperator`:
  https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/function/UnaryOperator.html)

Hinweis

- Sorgen Sie ggf. vorher dafür, dass Sie eine angemessene Projektstruktur haben.
- Passen Sie ggf. die `pom.xml` von ihren anderen Projekten an.

Warteschlangen

Eine Warteschlange ist eine einfache Datenstruktur bei der die Elemente in der Reihenfolge entfernt werden in der diese hinzugefügt wurden (🇺🇸 *First-in-First-out (FiFo)*).

Zentrale Methoden einer Warteschlange `Queue<T>` sind:

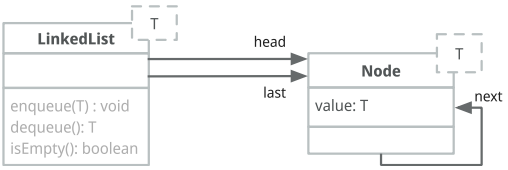
- `void enqueue(T item)` (auch `boolean offer(T item)` bzw. `void add(T item)`): Fügt ein Element hinzu.
- `T dequeue()` oder `T poll()`: Entfernt das älteste Element und gibt es zurück.
- `boolean isEmpty()`: Gibt an ob die Warteschlange leer ist.

Verkettete Listen

Zur Erinnerung

Die Verwendung eines Arrays, um Listen zu speichern, hat den Nachteil, dass sie eine feste Größe haben und nicht effizient vergrößert werden können.

Eine verkettete Liste ist eine alternative Implementierungstechnik, die flexibler ist als ein Array und dynamisch wachsen kann. Jedes Element in einer verketteten Liste enthält eine Referenz auf das nächste Element in der Liste.



Wann sollte welche Implementierung für eine Liste verwendet werden?

Sind die minimale und die maximale Größe einer Liste sehr gut abschätzbar, und liegen diese beiden Werte *hinreichend* nah beieinander, dann ist die Verwendung eines Arrays allein aus speichertechnischer Sicht effizienter als eine verkettete Liste, da der Overhead pro Element geringer ist. Weiterhin ist der Zugriff auf die Elemente effizienter.

Sollten jedoch auch Operationen wie Einfügen oder Löschen *häufig* vorkommen - insbesondere wenn diese nicht am Ende der Liste vorgenommen werden - ist eine detaillierte Analyse notwendig. Im Zweifelsfall ist meist eine Arraylist die bessere Wahl, da die Cache-Effizienz ggf. besser ist und auch der wahlfreie Zugriff auf die Elemente effizienter ist.

Java `Optionals`

Instanzen der Klasse *`java.util.`*`Optional<T>` (bzw. *`java.util.`*`OptionalInt` etc.) *repräsentieren Werte die vorhanden sind oder auch nicht*.

Insbesondere *`java.util.`*`Optional<T>` kann/sollte anstelle von `null` verwendet werden, in Fällen in denen unter bestimmten Umständen kein sinnvoller Wert angegeben werden kann.

Bemerkung

Es gibt moderne Programmiersprachen, die auf `null` komplett verzichten und stattdessen immer auf `Optionals` oder ähnliche Konstrukte setzen.

Beispiel

```
static Optional<Integer> min(int[] a ) {
    if(a == null || a.length == 0)
        return Optional.empty();

    int min = a[0];
    for(int x: a) { if (x < min) { min = x; } }
    return Optional.of(min);
}
```

✎ Übung

2.3. Implementierung einer Warteschlange mittels verketteter Liste

Implementieren Sie eine Warteschlange (`Queue<T>`) basierend auf einer verketteten Liste. Ihre Klasse `LinkedQueue<T>` soll das folgende Interface `Queue<T>` implementieren.

```
1  public interface Queue<T> {
2      void enqueue(T item);
3      Optional<T> dequeue();
4      boolean isEmpty();
5      int size();
6
7      void replaceAll(UnaryOperator<T> operator);
8      void forEach(Consumer<T> operator);
9      <X> Queue<X> map(Function<T, X> mapper);
10     static <T> Queue<T> empty() { TODO }
11 }
```

Erklärungen

`map`:	Erzeugt eine neue `Queue<X>` bei der die Elemente der neuen Queue das Ergebnis der Anwendung der Funktion `apply` des Objekts `mappers` auf die Element der `Queue` sind.
`empty`:	Erzeugt eine leere Queue.

Schreiben Sie Testfälle, um die Implementierung zu überprüfen. Zielen Sie auf mind. 100% *Statementcoverage* ab.

▤ Zusammenfassung

Funktionale Programmierung

- Lambdas sind Ausdrücke, die (anonyme) Funktionen repräsentieren.
- Es sind sowohl Referenzen auf statische Methoden als auch Instanzmethoden und sogar Konstruktoren möglich.
- Currying wird nicht direkt unterstützt, aber durch die Verwendung von Funktionskomposition und partieller Anwendung kann es simuliert werden.
- Anstelle von `null` als Rückgabewert sollte man `Optional<T>` bevorzugen, um klar zu kommunizieren, dass es auch keinen Rückgabewert geben kann.